EECS 280 - Lecture 2

Procedural Abstraction

https://eecs280staff.github.io/notes/02_ProceduralAbstraction_Testing.html

Announcements

- P1 due 1/19 (little over a week)
  - Will discuss a bit today
- Remember to fill stuff out!
  - CARES survey (0.5% of your total grade) by 1/26
  - Coaching form (Piazza @105)
  - Exam accommodations form
  - Exam conflict forms 1/28
- Labs start this week
- Office hours in full swing – see website

Interested in Research?

Q&A and Polls During Lecture

- If you have questions during lecture, please raise your hand!
- I'll also keep an eye on this Q&A link
- I will also use this for live polls

Yo, Teach!

- "If you did CS here, you must have some good tips for 280! How do I make sure I get the best I can?"
  - Come to office hours! (Both staff and Professor… it's my favorite part of the job!)
    - Early in the project cycle, before they get busy
  - Get started with projects early
  - Test your project as you design it (more next lecture)

Last Time

- Class Introduction
- Talked about the difference between compile time and run time
  - compile time: the static code we write
  - run time: the dynamic execution of the code we execute
- Reference variables are extra names we give to preexisting objects

Last Time

- With reference semantics, variables are aliases
- In C++, declare a **reference** using the `&` qualifier
- Declaring a reference does ***not*** create a new object
- Instead, it binds a **new name** to an existing object!

```
int x = 42;    //initialize value of x to 42
int z = 3;     //initialize value of z to 3
int& y = x;  //y and x are two names for one object
x = 24;       //assigns 24 to object named x, AKA y
y = z;        //Does NOT bind y to different object
              //Value semantics used here (both y and x update to 3)
```

Memory diagram

Some lingering questions…

- *“If I took 183 do I need to do any of the set up? Do I need to do Homebrew? I'm having a ton of trouble with it?"*

- *"Is this class a standalone c++ class from the ground up, or are we expected to have prior knowledge from other classes like eecs 183?"*

Today

- What is a **call stack**?
  - How does it behave?
- Procedural Abstraction
  - How writing **good** functions makes your code much easier to maintain
  - Some tips for P1

Agenda

- **Wrap up Machine Model**
- Function Calls and the Call Stack
- Procedural Abstraction

Is this a compile-time error or runtime?

```
int main() {
  int x = 0;
  int y = 3 / x;
  return 0;
}
```

Dividing by zero… uh-oh

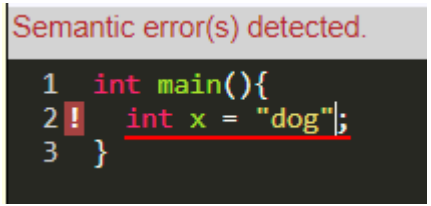
Poll: Will this cause an error during compilation or during execution?

variable doesn't have value ; object has value

Compile time vs. run time

↓ data type problem

- **Values** really only have meaning during execution
- Errors based on values usually happen at run time
- Examples of compile time errors are things like invalid syntax, or incompatible data types



values don't exist until we run the program.

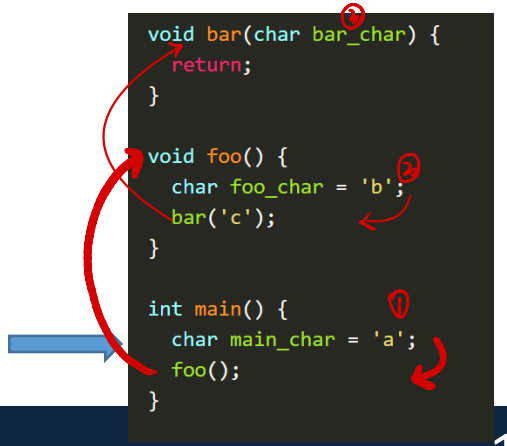
Agenda

- Wrap up Machine Model
- **Function Calls and the Call Stack**
- Procedural Abstraction

Function Calls

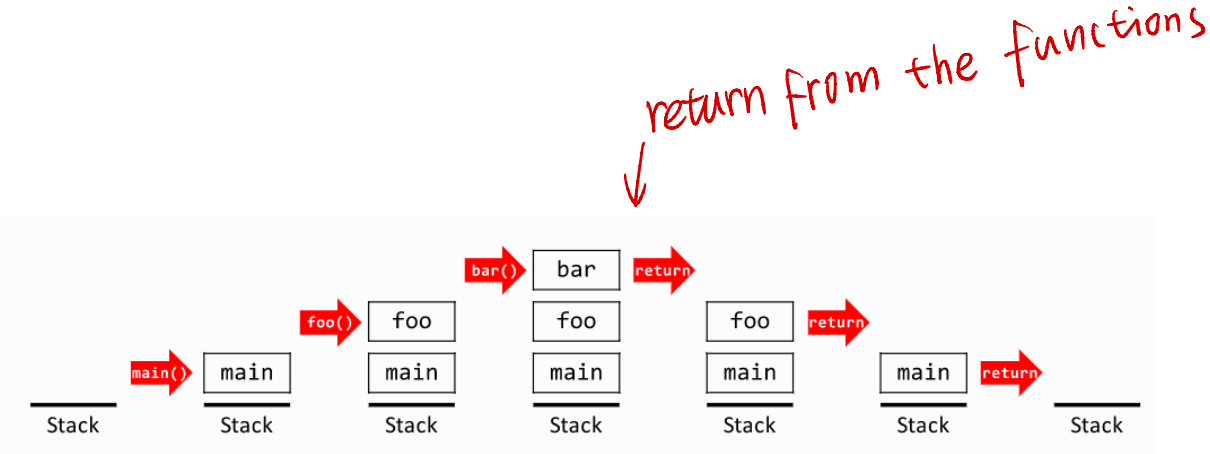
- All of a function's data (parameters, local variables, etc), get bundled in an **activation record** or **stack entry** and pushed on a "stack" structure in memory
  - Only the "top of the stack" is ever accessed

Notice all the addresses are contiguous



Function Calls

- As we call more and more functions, the stack gets larger
- As we return from functions, the stack gets smaller



Lobster Demo

- Step forward/backward
- Compile errors
  - Hover over ! to get error
  - underline compile errors
- Arrow keys go forward, backward,
- Arrows on side will jump to code

Lobster exercise: The Call Stack

the red number in stack is actually the old number of the last variable.

Poll: What is the maximum amount of memory used by this program at one time (assuming ints are 4 bytes)?

10 → 40 bytes

```
int min(int x, int y) {
  if (x < y) { return x; }
  else { return y; }
}

int minOf3(int x, int y, int z) {
  int a = min(x, y);
  int b = min(y, z);
  return min(a, b);
}
```

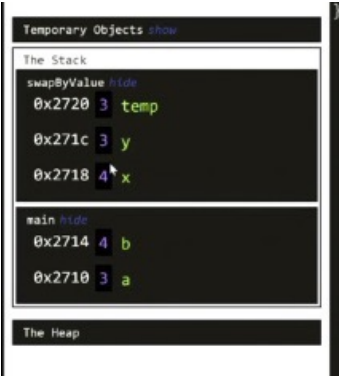
```
int main() {
  int a = 3;
  int b = 4;
  int c = 5;

  // prints 3
  cout << minOf3(a, b, c);
}
```

Try this yourself in L02.2_call_stack on Lobster!

Pass By Value

Poll: What does the second print statement output?

- Regular parameter passing is done **by value**.
  - i.e. We don't pass objects, just their values!




```
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b;
  swap(a, b);
  cout << a << ", " << b;
}
```

3 7

It doesn't do anything.

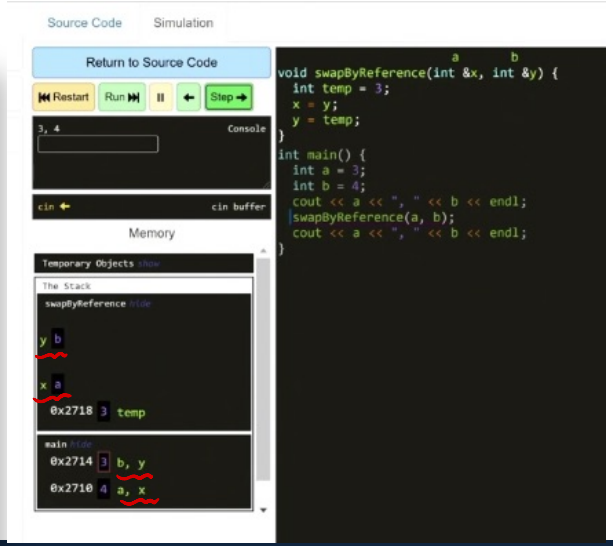
Pass By Reference

not creating a new object

(or new memory location)

- You can also pass **by reference**.
  - i.e. The parameter is just another name (an alias) for the original object. It's like passing the whole object.

```
void swap(int &x,int &y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b;
  swap(a, b);
  cout << a << ", " << b;
}
```



Agenda

- Wrap up Machine Model
- Function Calls and the Call Stack
- **Procedural Abstraction**

Procedural Abstraction

- Or… why the heck are there so many files?!

- In general…
  - …helps manage complexity.
  - …hides details.

- Most important words
  - what
  - how ← hide this since it is complicated.

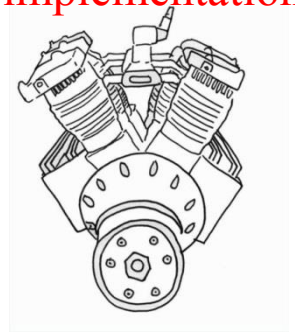
Example from Last Time

What

Simple Abstraction

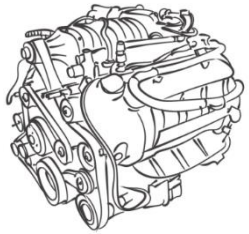
How

Complex implementation



Rotate wheel clockwise to turn right, push right pedal to accelerate…

Alternate implementation



Project 1

- Objectives:
  - Implement several functions to calculate meaningful statistics on a vector of data
    - E.g. count, median, standard deviation
    - A vector is a fancy way to list several numbers (see appendix B for examples)
  - Design a “driver” program that can read a bunch of data from a spreadsheet, store it in vectors, and call several of the above functions

Project 1 - Driver

- Thankfully, we already have a function that can extract data from a spreadsheet!
- It's in p1_library.cpp:

```
//EFFECTS: extracts one column of data from a tab separated values file (.tsv)
//  Prints errors to stdout and exits with non-zero status on errors
std::vector<double> extract_column(std::string filename,
                                   std::string column_name) {
```

- Question: what's the best way to make use of this function?

Project 1 - Driver

```
int main() {
// ...
V = extract_column(f, c);
```

it doesn't work. since doesn't show before in this file.

```
//EFFECTS: extracts one column of data from a tab separated values file (.tsv)
//  Prints errors to stdout and exits with non-zero status on errors
std::vector<double> extract_column(std::string filename,
                                   std::string column_name) {
```

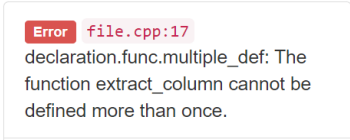
- Question: what's the best way to make use of this function?
- Should we… copy-and-paste `extract_column` into `main.cpp`?

Poll: Why is copy-pasting bad? (choose all that apply)

a) It makes updating code error-prone
b) Code is slower to execute
c) Code is harder to read
d) Might have compiler error when multiple functions have same name

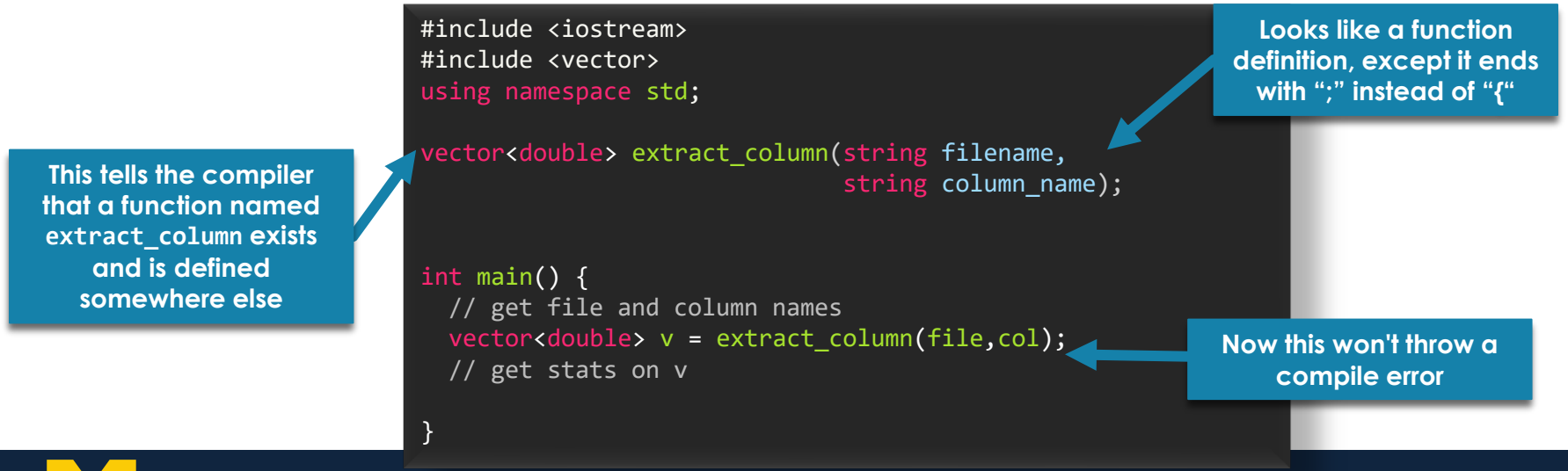
Don’t Copy-Paste Code!!

- Often have very useful functions we want to use in several places
  - E.g. extract_column, max, vector::push_back, printing to the terminal, etc
- Makes our code bloated and hard to read
- Copy-pasting this code means we have several copies
  - If we fix a bug, or make an improvement, we must update **all of them**
  - Error prone, we’ll probably miss at least one
- If we compile multiple files that have copies of the same function, we'll get a compiler error due to duplicate names



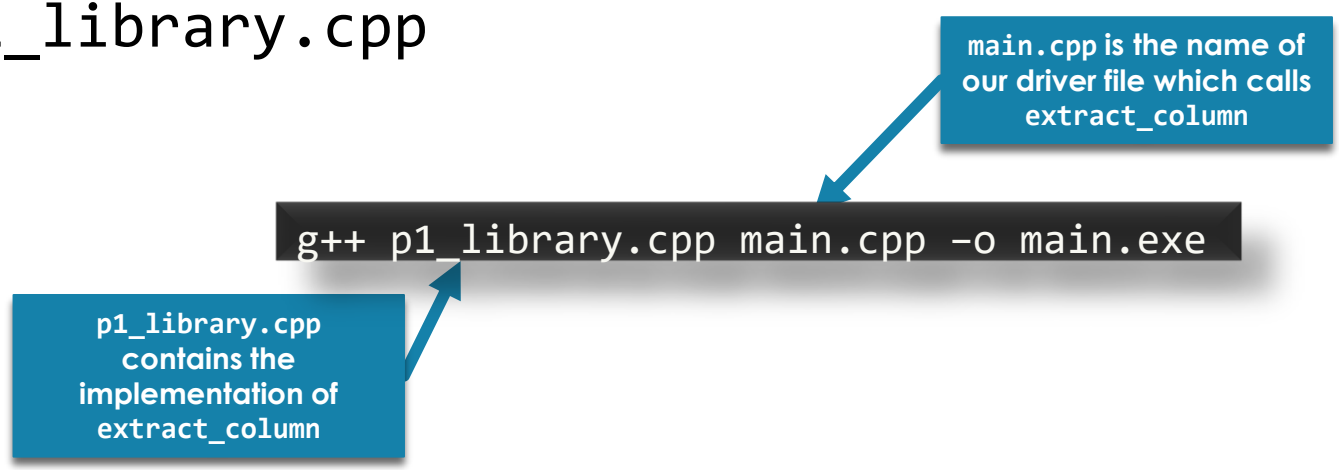
Partial Improvement

- Let's leave our function where it is
  - But we'll include a "forward declaration"



Partial Improvement

- Then, when compiling, we will also pass `p1_library.cpp`



Partial Improvement

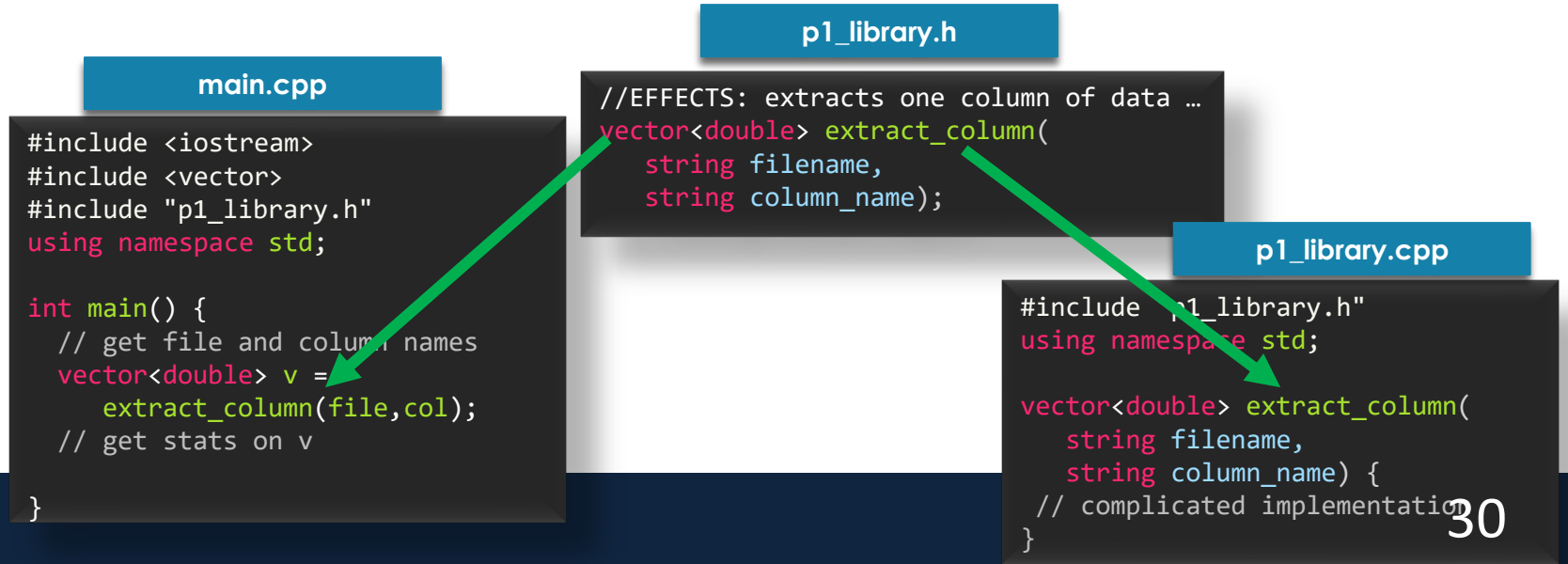
- This is better, but we still might have tons of forward declarations at the start of each file

These could include 100s or 1000s of functions in a realistic program!

```
#include <iostream>
#include <vector>
using namespace std;

// functions from p1_library.cpp
vector<double> extract_column(string filename,
                              string column_name);
// functions from stats.cpp
double mean (vector<double> v);
double sum  (vector<double> v);
int    count(vector<double> v);

int main() {
  // get file and column names
  vector<double> v = extract_column(file,col);
  // get stats on v

}
```

Better Idea

- If we have a library with functions we'll use elsewhere…
  - …let's create a separate "header" file, which includes the declarations and short descriptions of what they do
  - "#include" the .h file in whatever files use it



Project 1 File Structure

- Add a header (`.h`) file for each module (except the driver)
- Here's an example using `stats`

The header contains all relevant declarations.

Stats.h

```
#include <vector>

double mean(std::vector<double> v);
```

```
#include <iostream>
#include <vector>
#include "stats.h"
using namespace std;

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}
```

Give both files to g++.

```
#include <vector>
#include "stats.h"
using namespace std;

double mean(vector<double> v) {
  // implementation not shown
}
```

Stats.cpp

```
g++ main.cpp stats.cpp -o main.exe
```

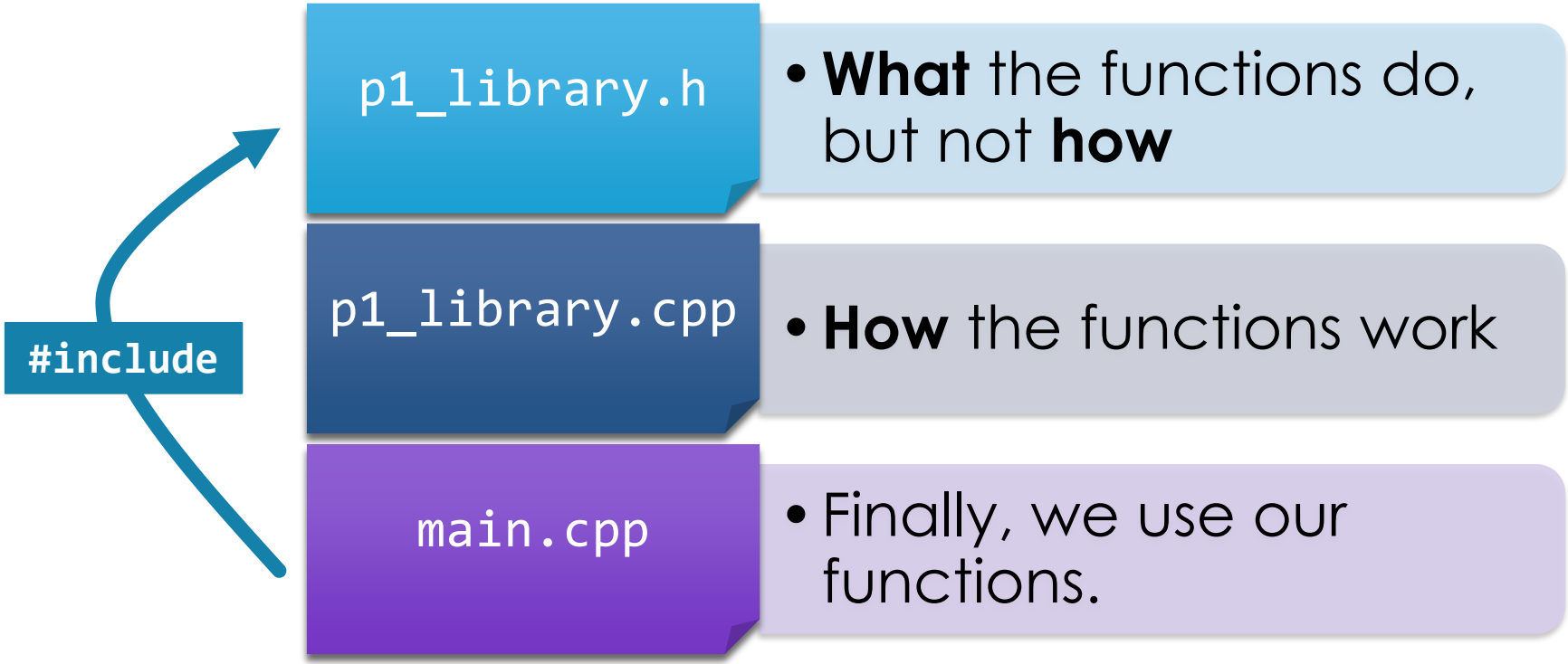
only include .h file

never include .cpp file

↓

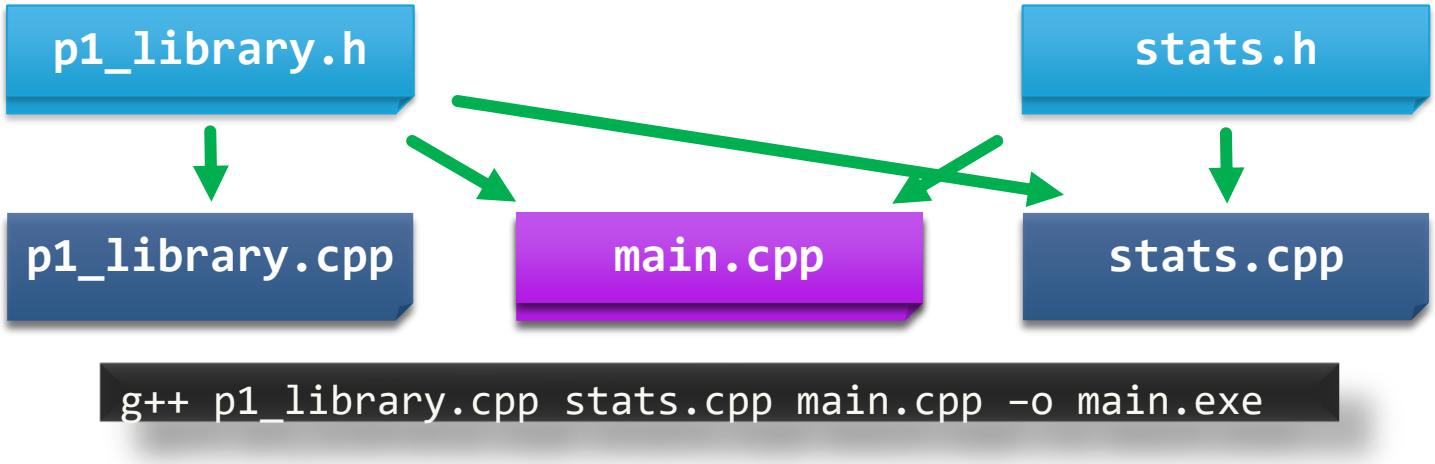
duplicate the implementation of the function.

Abstraction in Project 1



Project 1 File Structure

- Here's the overall structure.
  - Caution! Never `#include` a `.cpp` file.
  - Caution! Never put a `.h` file in the `g++` command.



Specification Comments

- It's a lot easier to read maintain code if all the developers use a consistent method for commenting and documenting
  - In this class, we use **RME** documentation
  - For every function, specify: **R**equires - **M**odifies - **E**ffects

RME Example

```
// REQUIRES: b != 0
//
// MODIFIES: c
//
// EFFECTS:  Divides a by b and stores result in
//           c.
int divide(int a, int b, int &c) {
  c = a / b;
}
```

we don't care about local variable.
only about any variable outside of the scope of this function.

Exercise: RME

```
// REQUIRES: ???
//
// MODIFIES: ???
//
// EFFECTS:  ???
//
void mystery(vector<int> &v) {
  for (int i = 0; i < v.size(); i++) {
    v[i] = v[i] / v.size();
  }
}
```

Which RME clauses can be omitted for this function?

A) none
B) REQUIRES also v.size() can be 0.
C) MODIFIES but if v.size()=0, the for loop doesn't work.
D) REQUIRES and MODIFIES
E) REQUIRES, MODIFIES, EFFECTS

Checking the **REQUIRES** Clause?

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  if (v.empty()) {
    // try to salvage the situation
  }
}
```

Don’t do this.

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  assert(!v.empty()); // sound the alarms!
}
```

Do this.

assert([EXPRESSION])

- assert() is a programmer's friend for debugging
- Does nothing if EXPRESSION is true
- Exits and prints an error message if EXPRESSION is false

```
#include <cassert>
int main() {
  int x = 3;
  int y = 4;
  assert(x < y); // ok, does nothing
  assert(x > y); // crash with debug message
}
```

```
$ ./test
Assertion failed: (false), function main,
file test.cpp, line 6.
```

Properties of Procedural Abstraction

- **Local**
  The implementation of an abstraction can be understood without examining any other abstraction implementation.

- **Substitutable**
  You can replace one (correct) implementation of an abstraction with another (correct) one, without having to change the way the abstraction is used.

  Separation of interface from implementation:
  Only depend on **interface**, not **implementation**!

Substitutability Example

- Here's the current implementation in `p1_library.cpp`:

```
void sort(std::vector<double> &v) {
  std::sort(v.begin(), v.end());
}
```

- And let's say your `mode` function in `stats.cpp` uses `sort`:

```
double mode(vector<double> v) {
  assert(!v.empty());
  sort(v);
  //...
}
```

- If the staff changes the implementation of `sort()`, do you need to change your mode function? No!

Next Time

- Discussion on testing
- Pointers!
  - A.k.a, The most powerful new tool over Eng 101 / EECS 183
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah